Code verification

CSE 331
University of Washington

Michael Ernst
Specification and verification

To find an error, compare two things

Program

Verification

Testing

Code review

Specification

Mental model

Example input & output

A form of specification!

How do you know that the test suite conforms to the spec?
Comparing a program to a specification

• Every behavior exhibited by the program is permitted by the specification
• Dynamic analysis = run the program (e.g., testing)
• Static analysis = don’t run the program (e.g., type checking)

• Problem: how to determine facts about all possible executions?
  • Dynamic analysis:
  • Static analysis:
Comparing a program to a specification

• Every behavior exhibited by the program is permitted by the specification
• **Dynamic analysis** = run the program (e.g., testing)
• **Static analysis** = don’t run the program (e.g., type checking)

Problem: how to determine facts about all possible executions?
• Dynamic analysis: **not possible**
• Static analysis: **estimate** what the program might do at run time
  • Execution: consider *both* branches of a conditional
  • Values: consider the *set* of values a variable might contain
A type is a set of values

- A type is a set of values
  - \texttt{int} contains 0, 1, 2, ...
  - \texttt{Integer} contains 0, 1, 2, ..., null
  - \texttt{String} contains "Hello World", "CSE 331", ",", null

- Some types have subset relationships
Type-checking is formal verification

- A type is a specification: what values are intended/expected
- The type-checker rejects the program if it cannot prove that the code meets the specification
- The type-checker does static analysis:
  - Consider all possible paths through the program
  - Consider sets of possible values for each variable
- Guarantee: the run-time value is in the set
  - The type is a trustworthy over-estimate
  - Virtual machine integrity
  - Detects/prevents programmer errors
Java's type system is too weak

Type checking prevents many errors

```java
int i = "hello";
```

Type checking doesn't prevent enough errors

```java
System.console().readLine();
```

```
NullPointerException
```
Motivation

java.lang.NullPointerException
Null pointer exception

Where is the defect? (Whose fault: implementer or client?)

String op(Data in) {
    return "transform: " + in.getF();
}
...
String s = op(null);
Null pointer exception

Where is the defect? (Whose fault: implementer or client?)

String `op(Data in)` {
    return "transform: " + in.getF();
}

... Can’t decide without a specification!

String `s = op(null);`
Specification 1: non-null parameter

```java
String op(@NonNull Data in) {
    return "transform: " + in.getF();
}
...
String s = op(null);
```
**Specification 1: non-null parameter**

```java
String op(@NonNull Data in) {
    return "transform: " + in.getF();
}
...
String s = op(null);
```

**Defect**
Specification 2: nullable parameter

```java
String op(@Nullable Data in) {
    return "transform: " + in.getF();
}
...
String s = op(null);
```
Specification 2: nullable parameter

String op(@Nullable Data in) {
    return "transform: " + in.getF();
}
...
String s = op(null);
Nullness Checker demo

- Programs to verify:
  - The Nullness Checker
  - JUnit 4.3

- Features:
  - Detect errors
  - Guarantee the absence of errors
  - Flow-sensitive type refinement
Type Checking

Source → Compiler → Executable

Errors:
- Fix bugs
- Change types

No errors
Optional Type Checking

Source → Compiler → No errors → Executable

Errors: Fix bugs, Change types

Warnings: Fix bugs, Add/change annotations

Optional Type Checker: Guaranteed behavior
Optional Type Checking

Source ➔ Compiler ➔ Executable

- No errors
- Fix bugs
- Change types

Errors ➔ Optional Type Checker ➔ Warnings

- Guaranteed behavior
- Fix bugs
- Add/change annotations

Source ➔ Compiler ➔ Executable

- No errors
- Fix bugs
- Change types
Benefits of type systems

- **Find bugs** in programs
  - Guarantee the **absence of errors**
- **Improve documentation**
  - Improve code structure & maintainability
- Aid compilers, optimizers, and analysis tools
  - E.g., could reduce number of run-time checks

- Possible negatives:
  - Must write the types (or use type inference)
  - False positives are possible (can be suppressed)
### Comparison: other nullness tools

<table>
<thead>
<tr>
<th></th>
<th>Null pointer errors</th>
<th>False warnings</th>
<th>Annotations written</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Found</td>
<td>Missed</td>
<td></td>
</tr>
<tr>
<td>Checker Framework</td>
<td>9</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>FindBugs</td>
<td>0</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>Jlint</td>
<td>0</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>PMD</td>
<td>0</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>Eclipse, in 2017</td>
<td>0</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>Intellij (@NotNull default), in 2017</td>
<td>0</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>6</td>
<td>1</td>
</tr>
</tbody>
</table>

Checking the Lookup program for file system searching (4kLOC)
Preventing null-pointer exceptions

Basic type system:
- @Nullable might be null
- @NonNull definitely not null

Default is @NonNull (opposite of Java’s default)
- Requires fewer annotations
- Makes the dangerous case explicit

(Nearly) no annotations in method bodies!
- Needed for some type arguments, as in List<@Nullable String>
Flow-sensitive type refinement

```java
if (myField != null) {
    myField.hashCode();
}

No need to declare a new local variable
```
One check for null is not enough

if (myField != null) {
    method1();
    myField.hashCode();
}

if (method2() != null) {
    method2().hashCode();
    myField.hashCode();
}

3 ways to express persistence across side effects:

@SideEffectFree void method1() { ... }
@MonotonicNonNull myField;
@EnsuresNonNull("myField") method1() {...}
Side effects (method, not type, annotations)

@SideEffectFree
   Does not modify externally-visible state

@Deterministic
   If called with == args again, gives == result

@Pure
   Both side-effect-free and deterministic

The side-effect annotations are trusted, not checked
Lazy initialization and persistence across side effects

@MonotonicNonNull type annotation, written on a field type

Might be null or non-null
May only be (re-)assigned a non-null value

Purpose: avoid re-checking
  Once non-null, always non-null
Example: Singleton pattern
Method pre- and post-conditions

Preconditions:
@RequiresNonNull

Postconditions:
@EnsuresNonNull
@EnsuresNonNullIf
   @EnsuresNonNullIf(expression="#1", result=true)
public boolean equals(@Nullable Object obj) { ... }
Polymorphism over qualifiers

/** Interns a String, and handles null. */
@PolyNull String intern(@PolyNull String a) {
    if (a == null) {
        return null;
    }
    return a.intern();
}

Like defining two overloaded methods:

@NonNull String intern(@NonNull String a) {...}
@Nullable String intern(@Nullable String a) {...}
A non-null field might contain null!

@NonNull String name;

MyClass() { // constructor
    ... this.name.hashCode() ... 
}

Initialization

@Initialized (constructor has completed)
@UnderInitialization(Frame.class)
    Its constructor is currently executing
@UnknownInitialization(Frame.class)
    Might be initialized or under initialization
Map keys and `Map.get`

```java
Map<String, @NonNull Integer> gifts;
... gifts.get("pipers piping").intValue() ...
```

Map.get can return null!

The Nullness Checker must treat any `Map.get()` as `@Nullable` ... unless

- the value type is non-null, and
- the argument key appears in the map.

Expressed with `@NonNull`
@KeyFor denotes a set of values

@KeyFor("myMap") String v; means v is a key in myMap

If myMap = { "red": "valor", "blue": "mystic", "yellow": "instinct" }
then @KeyFor("myMap") denotes the set { "red", "blue", "yellow" }

v = "red"   v = "blue"   v = "purple"   v = "mystic"   v = null

If myMap = { "bert": "tall", "ernie": "short" }
then @KeyFor("myMap") denotes the set { "bert", "ernie" }

v = "ernie"   v = "bert"   v = "red"   v = "mystic"   v = null

Assignments to myMap and v must maintain their relationship
Map key example

/** Computes predominators for each node in the graph. */
<T> Map<T, List<T>>
dominators(Map<T, List<KeyFor("#1") T>> predecessors) {
    ...
    for (T node : predecessors.keySet()) {
        for (T pred : predecessors.get(node)) {   // no NPE
            ... predecessors.get(pred) ...          // no NPE
        }
    }
}
Suppressing warnings

Because of Nullness Checker false positives

```java
if (x != null)
    // y has same nullness as x, which was just checked
    @SuppressWarnings("nullness")
    int z = y.field;

assert x != null : "@AssumeAssertion(nullness): ...";
```

More: https://checkerframework.org/manual/#suppressing-warnings

Write the rationale as a comment

Use smallest possible scope (e.g., local var)
Type-checking is modular

• Modular analysis = one procedure at a time
  • Contrast: whole-program analysis (slower, more precise)

• When analyzing a procedure, examines the specifications of callees
  • Never examines their implementation

```java
void client() {
    Object k = callee();
    myMap.get(k).toString();
}

Object callee() {
    Object k = ...;
    myMap.put(k, ...);
    return k;
}
```

```java
void client() {
    Object k = callee();
    myMap.get(k).toString();
}

@KeyFor("myMap") Object callee() {
    Object k = ...;
    myMap.put(k, ...);
    return k;
}
```

• Modular analysis = one procedure at a time

• Contrast: whole-program analysis (slower, more precise)

• When analyzing a procedure, examines the specifications of callees

  • Never examines their implementation

Possible Null-PointerException
Annotating external libraries

When type-checking clients, need library specification
The Nullness Checker comes with annotations for some libraries
For others, need to write specifications (or suppress warnings)
Two syntaxes:
  - As separate text file (stub file)
  - Within its .jar file (from annotated partial source code)
Checkers are usable

- Type-checking is familiar to programmers
- Modular: fast, incremental, partial programs
- Annotations are not too verbose
  - @NonNull: 1 per 75 lines
  - @Interned: 124 annotations in 220 KLOC revealed 11 bugs
  - @Format: 107 annotations in 2.8 MLOC revealed 104 bugs
- Few false positives
- First-year CS majors preferred using checkers to not
- **Practical**: in use in Silicon Valley, on Wall Street, etc.
Example type systems

Null dereferences (@NonNull)
   >200 errors in Google Collections, javac, ...

Equality tests (@Interned)
   >200 problems in Xerces, Lucene, ...

Concurrency / locking (@GuardedBy)
   >500 errors in BitcoinJ, Derby, Guava, Tomcat, ...

Fake enumerations / typedefs (@Fenum)
   problems in Swing, JabRef
String type systems

Regular expression syntax (@Regex)
  56 errors in Apache, etc.; 200 annos required

printf format strings (@Format)
  104 errors, only 107 annotations required

Signature format (@FullyQualified)
  28 errors in OpenJDK, ASM, AFU

Compiler messages (@CompilerMessageKey)
  8 wrong keys in Checker Framework
Security type systems

Command injection vulnerabilities (@OsTrusted)
5 missing validations in Hadoop
Information flow privacy (@Source)
SPARTA detected malware in Android apps

You can write your own checker!
Tips for pluggable type-checking

- Start small:
  - Start by type-checking part of your code
  - Only type-check properties that matter to you
- Use subclasses (not type qualifiers) if possible
- Write the spec first (and think of it as a spec)
- Avoid complex, unsound code
  - Avoid warning suppressions when possible
  - Avoid raw types like List; use List<String>
Verification

- **Goal:** prove that no bug exists
- **Specifications:** user provides
- **False negatives:** none
- **False positives:** user suppresses warnings
- **Downside:** user burden

Bug-finding

- **Goal:** find some bugs at low cost
- **Specifications:** infer likely specs
- **False negatives:** acceptable
- **False positives:** heuristics focus on most important bugs
- **Downside:** missed bugs

Neither is “better”; each is appropriate in certain circumstances.
Pluggable type-checking improves code

A type of formal verification:
- Write specifications
- Automatically check them

Featureful, effective, easy to use, scalable
Prevent bugs at compile time
Nullness is just one example type system

http://CheckerFramework.org/